URL: http://www.elsevier.nl/locate/entcs/volume62.html 14 pages

X-KLAIM and KLAVA: Programming Mobile Code

Lorenzo Bettini, Rocco De Nicola, Rosario Pugliese

Dipartimento di Sistemi e Informatica, Università di Firenze
{bettini,denicola,pugliese}@dsi.unifi.it

Abstract

Highly distributed networks have now become a common infrastructure for a new kind of wide-area distributed applications whose key design principle is network awareness, namely the ability to deal with dynamic changes of the network environment. Network-aware computing has called for new programming languages that exploit the mobility paradigm as the basic interaction mechanism. In this paper we present the KLAIM (*Kernel Language for Agent Interaction and Mobility*) framework for programming mobile code applications, namely the X-KLAIM programming language and the Java-based run-time system KLAVA. In particular, we illustrate how KLAVA handles mobile code. Finally, an example is shown that is implemented using this framework.

1 Introduction

Highly distributed networks have now become a common infrastructure for many applications which use network facilities to access remote resources and services. The Internet and the World Wide Web have surely contributed to this growth, making a network connection available to everyone, not confined to research laboratories or large enterprizes. Both programmers and users have to deal with a new kind of wide-area distributed applications whose key design principle is *network awareness*, namely the ability to deal with dynamic changes of their network environment.

Network-aware computing has called for new programming languages and paradigms that support migratory applications as a new model of interactions among clients and servers. *Mobile code*, i.e. software that can be sent to remote sites and can be executed on arrival, has been advocated as the basic paradigm to support network-aware programming (see, e.g., [23,12]). In the literature the term *mobility* is used to denote different mechanisms, ranging from simple ones, which only supply the ability of downloading code for execution (e.g. [3]), to more sophisticated ones, which support migration of entire computations (e.g. [25,1,21]).

In this paper we present the framework, which relies on Java, for programming in KLAIM, where mobile code applications and their interaction strategies can be naturally programmed. KLAIM (*Kernel Language for Agent Interaction and Mobility*) [14] is an experimental kernel language specifically designed to program distributed systems composed of several components interacting through multiple tuple spaces and mobile code. X-KLAIM (*eXtended* KLAIM) [6] is an imperative programming language obtained by extending KLAIM with variable declarations, operations with time-out, assignments, conditionals, sequential and iterative process composition. The implementation of KLAIM consists of two layers:

- a Java package, called KLAVA, which contains all the classes that implement the X-KLAIM runtime system and operations;
- the X-KLAIM compiler that translates X-KLAIM programs into Java programs that use the package KLAVA.

The structure of the KLAIM framework is depicted in Figure 1. X-KLAIM and KLAVA are available on line at `http://music.dsi.unifi.it`. KLAVA is briefly described in [6] and presented in detail in [4,7].

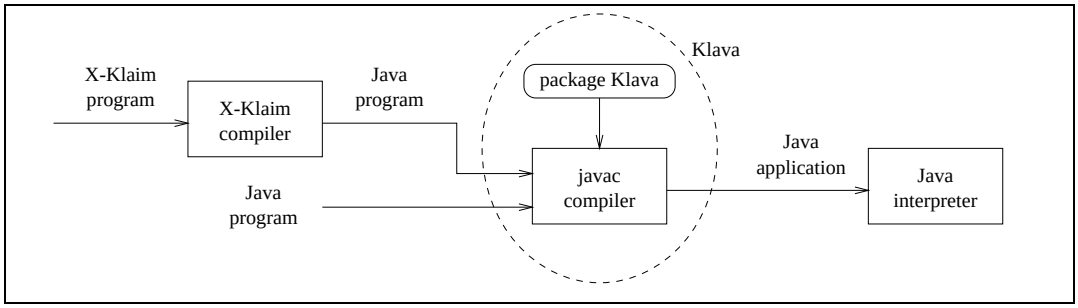

Fig. 1: The framework for X-KLAIM.

Let us briefly show how the framework can be used. If X-KLAIM source code is stored in a file called `foo.xklaim`, it can be compiled by means of the X-KLAIM compiler and the result will be the file `foo.java`. This last file can be compiled and executed by means of the standard jdk commands. When the program is executed, the host and the port number of the *Net server*, which is a class in the package KLAVA, must also be specified. This server keeps track of the physical localities of the nodes which are part of the net and must be started before any other node. Further details will be supplied in the next sections. Thus, X-KLAIM can be used to write the highest layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the KLAIM paradigm.

The rest of the paper is organized as follows. Section 2 introduces the language KLAIM and its implementation X-KLAIM. The Java package `Klava` and its general architecture are presented in Section 3, while the features specifically concerning code mobility are described in Section 4. In Section 5 we show how to program a simplified *news gatherer* that relies on mobile agents

for retrieving information on remote sites, in particular we will implement this example both in X-KLAIM and in KLAVA. Section 6 draws some conclusions and hints future work.

2 An overview of Klaim and X-Klaim

X-KLAIM (*eXtended* KLAIM) [6] is an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple tuple spaces and mobile code. It is based on the kernel language KLAIM [14] and is inspired by the coordination language Linda [18], hence it relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are containers of information items, called *fields*. There are two kinds of fields: *actual fields*, i.e. expressions, processes, localities, constants, identifiers, and *formal fields*, i.e. variables. Syntactically, a formal field is denoted with !*ide*, where *ide* is an identifier.

Tuples are anonymous and content-addressable and *pattern-matching* is used to select tuples in a tuple space. Two tuples match if they have the same number of fields and corresponding fields match: a formal field matches any value of the same type, and two actual fields match only if they are identical (but two formals never match). For instance, if *Val* is an integer variable, then tuples ("foo", "bar", !*Val*) and ("foo", "bar", 300) do match. After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *Val* will contain the integer value 300.

In Linda there is only one global shared tuple space; KLAIM extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. There are two kinds of localities: *physical localities* are the identifiers through which nodes can be uniquely identified within a net; *logical localities* are symbolic names for nodes. A reserved logical locality, `self`, can be used by processes to refer to their execution node. Physical localities have an absolute meaning within the net, while logical localities have a relative meaning depending on the node where they are interpreted and can be thought as aliases for network resources. Logical localities are associated to physical localities through *allocation environments*, represented as partial functions. Each node has its own environment that, in particular, associates `self` to the physical locality of the node.

KLAIM processes may run concurrently, both at the same node or at different nodes, and can execute the following operations over tuple spaces and nodes.

- **in**$(t)@l$: evaluates tuple t and looks for a matching tuple t' in the tuple space located at l. Whenever a matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is

found, the operation is suspended until one is available.

- **read**$(t)@l$: differs from **in**$(t)@l$ only because the tuple t' selected by pattern-matching is not removed from the tuple space located at l.
- **out**$(t)@l$: adds the tuple resulting from the evaluation of t to the tuple space located at l.
- **eval**$(P)@l$: spawns process P for execution at l.
- **newloc**(l): creates a new node in the net and binds its physical locality to l. The node can be considered as a "private" node because it can be accessed by the other nodes only if the creator communicates the value of variable l, which is the only way to access the fresh node.

During tuple evaluation, expressions are computed and logical localities are translated into physical ones. Evaluating a process implies substituting it with its *closure* (i.e. the process along with the environment of the node where the evaluation is taking place). This means that, e.g., **out**$(P)@l$ adds the closure of P to the tuple space located at l, while **eval**$(P)@l$ sends P, not its closure, for execution at l. Therefore, if node s_1 performs **out**$(P)@s_2$, then, when P is executed at s_2, `self` will actually refer to s_1. This means that *static scoping* is used. On the contrary, if s_1 performs **eval**$(P)@s_2$, no closure is sent: P will refer to s_2 when using `self` and *dynamic scoping* is used.

X-KLAIM extends KLAIM with a high level syntax for processes: it supplies variable declarations, operations enriched with *time-out*, assignments, conditionals, sequential and iterative process composition. Time-outs are added by supplying variants of the KLAIM blocking operations (**read** and **in**) that limit the waiting to a fixed amount of time. When operating over Wide Area Networks, this is necessary to deal with failures and performance degradation.

The X-KLAIM complete syntax can be found on-line, at the KLAIM site: `http://music.dsi.unifi.it`, while in Table 1 we report only the part concerning processes. We just briefly recall the more relevant features. If a timeout (expressed in milliseconds) is specified for an operation, through the keyword **within**, we get a boolean expression that can be tested in order to establish if the operation succeeded:

if in($!x$, $!y$)$@l$ **within** 2000 **then** ... *success!* **else** ... *timeout occurred* **endif**

Comments start with the symbol #, and local variables of processes are declared in the **declare** section of the process definition. Standard base types are available (**str**, **int**, etc.) as well as X-KLAIM typical types, such as **loc** for locality variables, **process** for process variables and **ts**, i.e. tuple space, for implementing data structures by means of tuple spaces, e.g. lists, that can be accessed through standard tuple space operations. Logical localities are declared by using the type **locname**.

I/O operations in X-KLAIM are implemented as tuple space operations. For instance the logical locality *screen* is actually attached to the output device. Hence, operation **out**("foo\n")$@$*screen* displays the string "foo\n"

RecProcDefs	::=	**rec** id formalparams procbody
	::=	**rec** id formalparams **extern**
	|	RecProcDefs **;** RecProcDefs
formalParams	::=	ϵ | [paramlist]
paramlist	::=	ϵ | id **:** type | paramlist **,** paramlist
procbody	::=	declpart **begin** proc **end**
declpart	::=	ϵ | **declare** decl
decl	::=	**const** id **:=** expression
	|	**locname** id
	|	**var** idlist **:** type
	|	decl **,** decl
idlist	::=	id | idlist **,** idlist
proc	::=	KAction | **nil**
	|	id **:=** expression | proc **;** proc
	|	**if** expression **then** proc **else** proc **endif**
	|	**while** expression **do** proc **enddo**
	|	**if** KAction **within** expression **then** proc **else** proc **endif**
	|	procCall | **call** id | **(** proc **)**
KAction	::=	**out(** tuple **)@**id | **in(** tuple **)@**id | **go@**id
	|	**eval(** proc **)@**id | **read(** tuple **)@**id | **newloc(** id **)**
tuple	::=	expression | proc | **!** id | tuple **,** tuple
procCall	::=	id **(** actuallist **)**
actuallist	::=	ϵ | expression | proc | id | actuallist **,** actuallist
id	::=	*string*
type	::=	**int** | **str** | **loc** | **process** | **ts** | **bool**

Table 1: X-KLAIM process syntax.

on the screen. X-KLAIM also supplies *strong mobility* by means of action **go**@l [5] that makes an agent migrate to l and resume its execution at l from the instruction following the migration.

The KLAIM and X-KLAIM Linda-like communication paradigm fits for coordinating heterogeneous, distributed and mobile applications. The underlying coordination model permits full *space uncoupling* of communicating objects because it requires a single interface: the operations over tuple spaces. This approach is also called *flow-of-objects* [2] as opposed to *method invocation*, which requires many interfaces for the operations supplied by remote objects. The Linda *asynchronous* communication model, known as *Generative Communication* [18], also permits *time uncoupling*, because it makes tuples' life time independent of the producer process' life time, and *destination uncoupling*, because the creator of a tuple is not required to know the future use or the destination of that tuple. Moreover, message selection is *associative* (by means of pattern matching) and *anonymous*, thus only the structure of a message has to be known.

3 The Klava package and its architecture

KLAVA (KLAIM *in Java*) is a Java package which contains all Java classes for implementing the runtime system support for X-KLAIM operations. KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the KLAIM paradigm.

Java [3] has been chosen as the implementation language for X-KLAIM because it supplies a natural support for programming distributed applications with mobile code. Indeed, Java supplies architectural independence, i.e. *on-line portability* [11], class libraries for network programming, tools for synchronization, dynamic class loading and customizable security mechanisms.

A KLAVA *net* is implemented by the KLAVA net server, which coordinates KLAVA nodes that are part of the net, by letting them communicate with each other. In order to become part of a KLAVA net, a node has to log into the net server by specifying its own physical locality, through which it will be uniquely identified and addressed within that net.

In the rest of this section, we shall present the "specializable" classes of the package `Klava`. Some of these classes can already be used as they are (e.g. class `Tuple`), while others have to be specialized through inheritance and methods overriding (e.g. class `KlavaProcess`).

The class `Tuple` provides methods for handling tuples (creating a tuple, adding elements to a tuple, getting an element of a tuple, etc.). A tuple can be created by passing a `Vector` object, containing all tuple elements, to the `Tuple` constructor, or by first creating an empty tuple and then adding elements using the method `add(Object o)`. To express a formal field, a `Class` object can be used. For instance, to create a tuple with a formal field of type `String` and an actual field of type `Integer` with value 10, one can write

```
Tuple t1 = new Tuple((new String()).getClass(), new Integer(10));
```

or

```
Tuple t1 = new Tuple(Class.forName("java.lang.String"), new Integer(10));
```

Another method of the class `Tuple` is `match`, that gets a tuple as parameter and checks the matching with the current tuple. For instance, the previous tuple matches the following one:

```
Tuple t2 = new Tuple(new String("Hello"), new Integer(10));
t2.match( t1 ) ;
```

`match` also performs the binding of the formals; after matching, the value bound to a formal field can be retrieved using the method `Object getItem(int index)`.

The interface `TupleItem` can also be used for handling tuple fields. Its methods are used by the matching algorithm: `isFormal` to test whether a tuple field is a formal, `setValue` to update a formal field with an actual value, and `equals` to test whether two actual fields match. As usual, the semantics of these methods must be specified by the classes that implement the interface. The package `Klava` makes available some classes for standard data types that implement this interface: `KString`, `KInteger`, `KBoolean` and `KVector`. It is assumed that an object of a class implementing `TupleItem` that has been created with the default constructor (i.e. with no parameters) is a formal. It is not necessary to use `getItem` to retrieve the value of formals: if `TupleItems`

are used as formal fields, fields values are automatically updated by means of the method `setValue`. Of course, since types are used for matching, a `KString` will never match a `String`.

The class `TupleSpace` provides methods to place tuples in and retrieve tuples from a tuple space. In particular, operations **out**, **in**, **read** and their non-blocking versions are implemented as methods of this class.

In KLAVA, localities (both logical and physical) are nothing but strings: the only Internet address that has to be known is the address (and port) of the host in which the net server is running. There are three classes that handle localities. The abstract class `Locality` is the base class. The other two classes `LogicalLocality` and `PhysicalLocality` are derived from this base class. A variable which represents a locality should always be declared as a `Locality` so that polymorphism can be used extensively. Physical localities are not IP addresses, but are simply the names with which `Node` objects register themselves into the `Net` object. In this way we have an additional abstraction level and a `Node` is independent from its IP address. All locality related classes implement the interface `TupleItem`, and thus localities can be used in tuples.

The class `Node` implements a node of a KLAVA net. A `Node` object contains a single tuple space and exports methods to access this tuple space. These methods will redirect operations to the corresponding methods of the tuple space of the node. The difference is that these methods also take a locality as parameter. Additionally, the class `Node` also provides the method `newloc`, that creates a new node in the net and returns its locality, and the method `eval`, that spawns a new process for execution. A `Node` object must log in a `Net` server, and hence has to know the exact IP address of the latter (host and port number). It must also specify its own physical locality; in case the proposed locality is already in the net, the net server will refuse registering the node. Two nodes can be started on the same machine, as long as they specify two different physical localities; indeed a physical locality is just a name, and not an Internet address. Every `Node` has two fields: `self` (of class `LogicalLocality`) and `here` (of class `PhysicalLocality`). `here` represents the physical locality of the node within the net. The environment of a `Node` can be specified with the method `void addToEnv(String logLoc, String phyLoc)`.

Due to network latency bandwidth, network communications can be quite slow, hence, retrieving information can require more time than one is willing to wait. Moreover, the absence of a tuple could block a process executing an **in**/**read** operation. To tackle these problems, a *time-out* can be used: if this expires before an operation returns, then a `KlavaTimeOutException` exception will be thrown. Time-outs can thus be handled in a `try...catch` block. For instance, a process can execute

```
try {
  in( s, loc1 , 5000 ) ;  // no more than 5 secs
  System.out.println( "I found " + s + " at locality " + loc1 ) ;
  ...
} catch ( KlavaTimeOutException toe ) {
  System.out.println( "TIME OUT!!!" ) ;
  ...
}
```

Nodes communicate through messages and streams (connected to sockets). The class `NodeMessage` implements messages exchanged in the KLAVA system (the content of a message can be any serializable `Object`). A message also contains the physical locality of the sender and of the receiver.

The class `KlavaProcess` is an abstract class that must be derived to create processes. The derived classes must implement the method `execute` that will be invoked when a process is executed (just like `run` for threads). A process must be executed within a node, which makes its *execution environment* [12]. To start a process within a new `Node`, one can invoke the method `addProcess(KlavaProcess P)`, as follows:

```
Node n = new Node( ... ) ;
n.start () ;
n.addProcess( new myProc() ) ;
```

`KlavaProcess` also offers all the methods to access tuple spaces; these methods transparently call the homonymous methods of the class `Node`. Even processes can own an allocation environment for localities. When a logical locality must be translated into a physical one, first the environment of the process (if it has one) is used, and then, if the interpretation fails, the environment of the node is used. Thus sending a closure (as explained in Section 2) consists in sending a process after setting its environment.

The class `Net` implements the server that manages a KLAIM net. A `Net` object keeps track of the physical localities of the nodes which are part of the net. It is a multithreaded server and can also be seen as a name registry server. There is exactly one net server for every KLAVA net. When a `Net` object receives a login request from a node, a new `NodeHandler` thread is spawned to handle the connection. `NodeHandler` will be a proxy for the node within the net and will handle the delivery of node's messages to other nodes; this form of inter node communication is depicted in Figure 2, where `Nodes` is a table mapping node physical localities into `NodeHandler`s.

In this scenario communications take place indirectly, through the net server. Direct connections are also allowed: a node can ask the net server for the IP address of another node and then it can establish a direct connection to that node; in this case messages are delivered to the receiving node directly. Note that in case of firewalls or network restrictions the access to a remote site may be allowed only through a net server: e.g., an applet, by default, can only open a network connection to the computer it has been downloaded

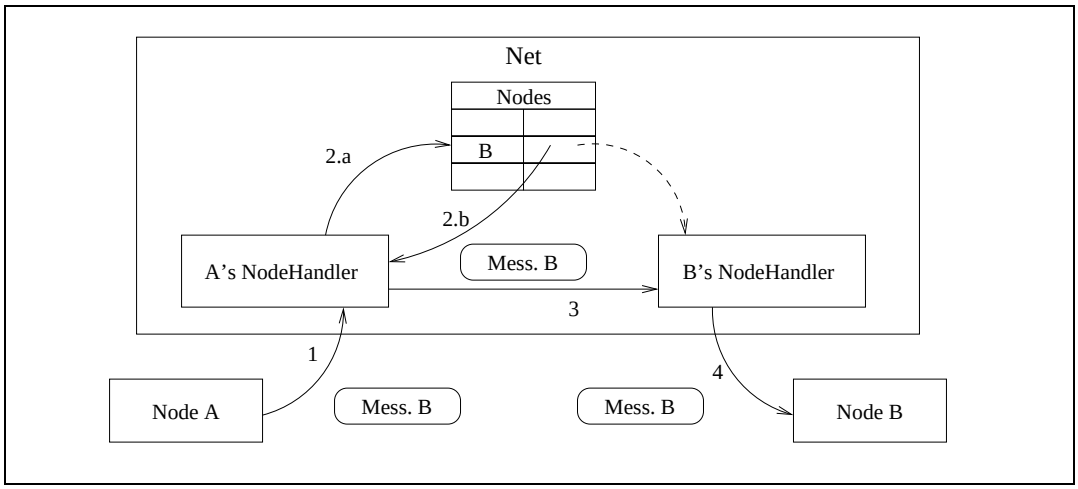

Fig. 2: Inter node communication through `NodeHandler`s.

from. If on the latter computer there's a net server running, the applet is still able to communicate, indirectly, to all the nodes and, possibly, applets that are part of that KLAVA net. An example of an applet built with KLAVA is available at `http://music.dsi.unifi.it/klava_applet`.

4 Code Mobility in Klava

Processes can be sent along with a message and executed at destination sites, where however their Java classes (i.e. their code) may be unknown. It is necessary to make such a code available for execution at remote hosts. Instead of an *on-demand* approach (where the code is requested to the server from which an agent is downloaded when it is needed), we prefer to collect all the code that a process needs, before dispatching it. This approach better complies with the mobile agents paradigm: during a migration, an agent will bring all the information that it may need for later executions. Moreover, our choice has the advantage of simplifying the handling of *disconnected operations* [22].

Therefore, a process must be sent along with its class binary code, and with the class code of all the objects the process uses. Clearly, only the code of user defined classes has to be sent, as the other code (e.g. Java and `Klava` classes) is common to every KLAVA application. The names of user defined classes can be retrieved by means of class introspection (*Java Reflection API*). Just before dispatching a process to a remote site, a recursive procedure is called for collecting all classes that are used by the process when declaring data members, objects returned by or passed to a method/constructor, exceptions thrown by methods, inner classes, the interfaces implemented by its class, the base class of its class. The byte code of these classes is then sent, along with the migrating process.

All the nodes that are willing to accept remote processes (due to security problems, a node may refuse accepting remote processes for execution) must have a custom *class loader*: a `NodeClassLoader`, provided in the `Klava` pack-

age. When a class code is needed, if the class loader cannot find the code among the local packages, it will try to find it in its own local table of class binary data. Therefore, when a process is received from the network, before using it, the node must add the class data (received along with the process) to its class loader's table.

Due to security concerns Java does not allow dynamic inspection of byte code stack; this makes impossible to save the execution state for later use. For this reason, KLAVA can only permit *weak mobility* of agents that have to be restarted after the migration, while X-KLAIM, by relying on a source level transformation [5], also provides *strong mobility* by means of **go**@l operation (the mobile agent automatically resumes execution from the point after the migration). A more detailed description of forms of mobility can be found in [13,20].

Downloading code from the net exposes the executing machine to security risks, since this code could execute dangerous operations that could damage the system or the other executing processes. `Klava` provides a `KlavaSecurityManager`, which, if activated by the node, does not allow processes, downloaded from the net, or sent by remote nodes, to execute operations on system resources (such as files, and system properties). We are going to implement new security mechanisms that exploit the new Java security model [19].

5 An example of mobility: a *News Gatherer*

In this section we will show how to program a *news gatherer*, that relies on mobile agents for retrieving information on remote sites, using our framework. In particular we will implement this example both in X-KLAIM and in KLAVA. We assume that some data are distributed over the nodes of a KLAVA net and that each node either contains the information we are searching for, or the locality of the next node to visit in the net. This example is taken from [14], and in KLAIM can be specified [1] as shown in Listing 1.

$$
\begin{array}{l}
\mathit{NewsGatherer}(\mathit{item}, \mathit{retLoc}) = \\
\quad \mathbf{read}(\mathit{item}, !\mathit{itemVal})@\mathtt{self}.\mathbf{out}(\mathit{itemVal})@\mathit{retLoc}.\mathbf{nil} \\
\quad + \\
\quad \mathbf{read}(\mathit{item}, !\mathit{nextLoc})@\mathtt{self}.\mathbf{eval}(\mathit{NewsGatherer}(\mathit{item}, \mathit{retLoc}))@\mathit{nextLoc}.\mathbf{nil}
\end{array}
$$

Listing 1: The news gatherer specified in KLAIM.

The agent *NewsGatherer* tries to read one of two possible tuples: the first tuple contains information we are searching for and the second one the locality of the next node to visit. In the first case, the agent communicates the result

[1] $P_1 + P_2$ spawns both processes P_1 and P_2, but only one of the two will continue its execution.

to its owner and terminates, in the other case it simply spawns itself to the next node (Figure 3).

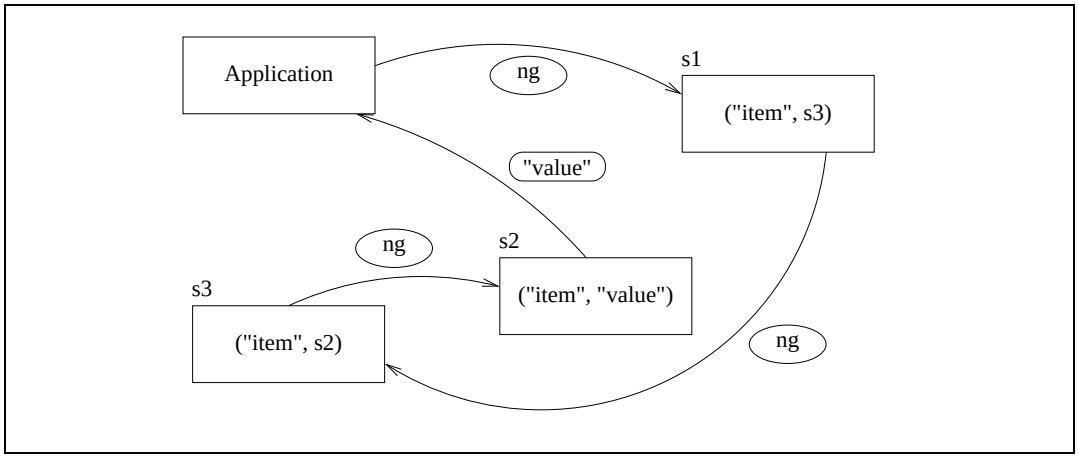

Fig. 3: The news gatherer example (ng is the news gatherer agent).

The implementations in X-KLAIM and in KLAVA are reported, respectively, in Listing 2 and 3. If the result of the query is a locality, a new instance of the process `NewsGatherer` is remotely spawned by means of an **eval**. We use a timeout to test the presence of the tuple containing the information: if this is not found within two seconds, we retrieve the locality of the next node to visit. Notice that, also in KLAVA, for spawning a new process on a remote site, only an **eval** invocation is required: the underline system will take care of serializing the process through the network together with code and the code of all the classes it will use and the values of its fields. The two implementations are quite similar, and indeed the code generated by the X-KLAIM compiler is not much different from the one shown in Listing 3.

```
rec NewsGatherer[ item : str, retLoc : loc ]
  declare
    var itemVal : str ;
    var nextLoc : loc
  begin
    if read( item, !itemVal )@self within 2000 then
      out( itemVal )@retLoc
    else
      read( item, !nextLoc )@self ;
      eval( NewsGatherer( item, retLoc ) )@nextLoc
    endif
  end
```

Listing 2: The implementation of the news gatherer in X-KLAIM.

In X-KLAIM strong mobility can be exploited, thus the same example can be implemented by using the **go**@l operation, as illustrated in Listing 4.

```
class NewsGatherer extends KlavaProcess {
  protected KString itemVal ;
  protected KString item ;
  protected Locality retLoc ;

  public NewsGatherer( KString item, Locality retLoc ) {
    this.item = item ;
    this.retLoc = retLoc ;
  }

  public void execute() throws KlavaException {
    itemVal = new KString();
    try {
      read( item, itemVal, self , 2000 );
      out( itemVal, retLoc );
    } catch (KlavaTimeOutException e) {
      Locality nextLoc = new PhysicalLocality();
      read( item, nextLoc, self );
      eval( new NewsGatherer( item, retLoc ), nextLoc );
    }
  }
}
```

Listing 3: The implementation of the news gatherer in KLAVA.

```
rec NewsGatherer[ item : str, retLoc : loc ]
  declare
    var itemVal : str ;
    var nextLoc : loc ;
    var found : bool
  begin
    found := false;
    while not found do
      if read( item, !itemVal )@self within 2000 then
        out( itemVal )@retLoc ;
        found := true ;
      else
        read( item, !nextLoc )@self ;
        go@nextLoc
      endif
    enddo
  end
```

Listing 4: X-KLAIM implementation exploiting strong mobility.

6 Conclusions and Future Work

We presented the KLAIM framework. The underlying programming model enables *space uncoupling*, *time uncoupling* and *destination uncoupling*, and *asynchronous*, *associative* and *anonymous* communication. We believe that the KLAIM framework is suitable for programming distributed applications, mobile agents, and, more in general, mobile code. An interesting spin-off of

our approach is that since it is based upon the KLAIM formal model, some properties of systems can be formally established. Indeed, a modal logic for KLAIM is being studied [17] and a system to automatically prove KLAIM system properties is under development.

A number of extensions have been made to the original KLAIM model of computation. In [15,16], KLAIM is extended with a capability-based type system that provides direct support for expressing and for using policies that control accesses to resources and data. In [8], KLAIM is enriched in order to transform the underlying flat model into a hierarchical model, that permits modelling structured nets, and in [9,10] node connectivity is made explicit. We plan to implement all these new features in the near future.

References

[1] Acharya, A., M. Ranganathan and J. Saltz, *Sumatra: A Language for Resource-aware Mobile Programs*, in: Vitek and Tschudin [24], pp. 111–130.

[2] Arnold, K., E. Freeman and S. Hupfer, "JavaSpaces Principles, Patterns and Practice," Addison-Wesley, 1999.

[3] Arnold, K., J. Gosling and D. Holmes, "The Java Programming Language," Addison-Wesley, 2000, 3rd edition.

[4] Bettini, L., "Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile," Master's thesis, Dip. di Sistemi e Informatica, Univ. di Firenze (1998).

[5] Bettini, L. and R. De Nicola, *Translating Strong Mobility into Weak Mobility*, in: *Proc. of the Fifth IEEE Int. Conf. on Mobile Agents (MA 2001)*, 2001, to appear.

[6] Bettini, L., R. De Nicola, G. Ferrari and R. Pugliese, *Interactive Mobile Agents in* X-KLAIM, in: P. Ciancarini and R. Tolksdorf, editors, *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (1998), pp. 110–115.

[7] Bettini, L., R. De Nicola and R. Pugliese, KLAVA*: a Java Framework for Distributed and Mobile Applications*, Tech. Report, Dip. Sistemi e Informatica, Univ. di Firenze (2001), available at `http://music.dsi.unifi.it/papers.html`.

[8] Bettini, L., M. Loreti and R. Pugliese, *Structured Nets in* KLAIM, in: *Proc. of ACM SAC 2000, Special Track on Coordination Models, Languages and Applications* (2000), pp. 174–180.

[9] Bettini, L., M. Loreti and R. Pugliese, *Modelling Node Connectivity in Dynamically Evolving Networks*, in: *Proc. of CONCOORD, Int. Workshop on Concurrency and Coordination*, ENTCS **54**, 2001.

[10] Bettini, L., M. Loreti and R. Pugliese, *An Infrastructure Language for Open Nets*, in: *Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications*, 2002, to appear.

[11] Cardelli, L., *Mobile computation*, in: Vitek and Tschudin [24], pp. 3–6.

[12] Carzaniga, A., G. Picco and G. Vigna, *Designing Distributed Applications with Mobile Code Paradigms*, in: R. Taylor, editor, *Proc. of the 19th Int. Conf. on Software Engineering (ICSE '97)* (1997), pp. 22–33.

[13] Cugola, G., C. Ghezzi, G. Picco and G. Vigna, *Analyzing Mobile Code Languages*, in: Vitek and Tschudin [24].

[14] De Nicola, R., G. Ferrari and R. Pugliese, KLAIM*: a Kernel Language for Agents Interaction and Mobility*, IEEE Transactions on Software Engineering **24** (1998), pp. 315–330.

[15] De Nicola, R., G. Ferrari and R. Pugliese, *Types as Specifications of Access Policies*, in: J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, number 1603 in LNCS (1999), pp. 117–146.

[16] De Nicola, R., G. Ferrari, R. Pugliese and B. Venneri, *Types for Access Control*, Theoretical Computer Science **240** (2000), pp. 215–254.

[17] De Nicola, R. and M. Loreti, *A Modal Logic for* KLAIM, in: T. Rus, editor, *Proc of Algebraic Methodology and Software Technology, 8th Int. Conf. AMAST 2000*, number 1816 in LNCS (2000), pp. 339–354.

[18] Gelernter, D., *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[19] Gong, L., "Inside Java 2 platform security: architecture, API design, and implementation," Addison-Wesley, Reading, MA, USA, 1999.

[20] Hohlfeld, M. and B. Yee, *How to Migrate Agents* (1998), available at `http://www.cs.ucsd.edu/~bsy`.

[21] Lange, D. and M. Oshima, "Programming and Deploying Java Mobile Agents with Aglets," Addison-Wesley, 1998.

[22] Park, A. and P. Reichl, *Personal Disconnected Operations with Mobile Agents*, in: *Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*, Tokyo, 1998.

[23] Thorn, T., *Programming Languages for Mobile Code*, ACM Computing Surveys **29** (1997), pp. 213–239, also Technical Report 1083, University of Rennes IRISA.

[24] Vitek, J. and C. Tschudin, editors, "Mobile Object Systems - Towards the Programmable Internet," Springer, 1997.

[25] White, J. E., *Mobile Agents*, in: J. Bradshaw, editor, *Software Agents* (1996).